coNCePTuaL: A Network Correctness and Performance Testing Language

Scott Pakin

CCS-3: Modeling, Algorithms, and Informatics Group
Computer and Computational Sciences (CCS) Division
Los Alamos National Laboratory
E-mail: pakin@lanl.gov

Abstract

This paper introduces a new, domain-specific specification language called coNCePTuaL. coNCePTuaL enables the expression of sophisticated communication benchmarks and network validation tests in comparatively few lines of code. Besides helping programmers save time writing and debugging code, coNCePTuaL addresses the important—but largely unrecognized—problem of benchmark opacity. *Benchmark opacity refers to the current impracticality of presenting performance measurements in a manner that promotes reproducibility and independent evaluation of the results. For example, stating that a performance graph was produced by a "bandwidth" test says nothing about whether that test measures the data rate during a round-trip transmission or the average data rate over a number of back-to-back unidirectional messages; whether the benchmark pre-registers buffers, sends warm-up messages, and/or pre-posts asynchronous receives before starting the clock; how many runs were performed and whether these were aggregated by taking the mean, median, or maximum; or, even whether a data unit such as "MB/s" indicates 10^6 or 2^{20} bytes per second.*

Because coNCePTuaL programs are terse, a benchmark's complete source code can be listed alongside performance results, making explicit all of the design decisions that went into the benchmark program. Because coNCePTuaL's grammar is English-like, coNCePTuaL programs can easily be understood by non-experts. And because coNCePTuaL is a high-level language, it can target a variety of messaging layers and networks, enabling fair and accurate performance comparisons.

1. Introduction

Communication benchmarks play an important role in the area of high-performance computing. They help provide insight into application performance; they enable performance comparisons among disparate networks; and, they can be fed into performance models which are used to predict application and system performance [3, 4, 7]. Large-scale, parallel applications can spend a significant fraction of their total execution time communicating. For instance, Kerbyson et al. show how two representative, large-scale ASCI applications—SAGE and Sweep3D—may spend 30–50% of their execution time communicating when run on thousands of processors [5]. As a result, it is crucial that low-level network performance be characterized before one can understand and/or predict application performance.

Although communication benchmarks are important, benchmarking methodology is fraught with inconsistent terminology, variegated metrics for reporting results, and implementation artifacts that greatly impact performance. As a consequence, reported results are rarely reproducible and may present misleading estimates of how well a network will perform when used by a running application. Consider, for example, a bandwidth benchmark, which purportedly measures data rate versus message size. Although a bandwidth benchmark is one of the most common communication benchmarks, it may actually be expressed in a variety of different ways, causing radically different bandwidths to be reported. For instance, the communication pattern may be "throughput style", in which Node A sends a number of back-to-back messages to Node B and stops the clock upon receiving a short acknowledgment message—or even a full-length message—from Node B. Or, the pattern may be "ping-pong style", in which the two nodes repeatedly exchange messages, stopping the clock after the last message is received. Figure 1 quantifies the performance difference between these two approaches on a cluster of 1 GHz Itanium 2-based nodes interconnected with a Quadrics QsNet network [13]: the throughput style reports numbers from 71% to 161% of those reported by the ping-pong style—a significant range of differences. Regardless of the communication style used, the benchmark may run for a fixed number of messages or a fixed length of time. The benchmark may perform a number of warm-up iterations before starting the clock. Messages may be sent synchronously or asynchronously. Asynchronous receives may

Proceedings of the 18th International Parallel and Distributed Processing Symposium (IPDPS'04)

0-7695-2132-0/04/$17.00 (C) 2004 IEEE

be posted before or after the clock starts ticking. Even given the same basic benchmark, results can be reported differently. The number of messages transmitted and the statistical metric applied (e.g., mean, median, or maximum) can vary from benchmarker to benchmarker. Even something as simple as the units used for the results—"MB/s" designating either 10^6 or 2^{20} bytes per second—can induce a 5% sway of the numbers.

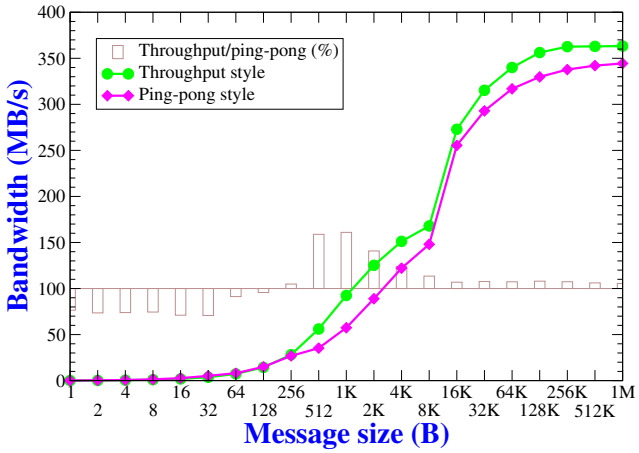


Figure 1. Relative performance of throughput vs. ping-pong bandwidth on an Itanium 2 + Quadrics cluster

Although there is often good reason for expressing a particular benchmark in one way or another, the real problem—which we call *benchmark opacity*—is that it is not obvious beyond a superficial notion as to what precisely the benchmark measures. Benchmark opacity leads to misinterpreted results, incorrectly drawn conclusions, and, eventually, to money being spent on a network that suboptimally handles important applications' communication needs. The ideal solution to the problem of benchmark opacity is to publish the complete benchmark source code alongside the corresponding measurement data. Unfortunately, this practice is largely impractical, as even a simple benchmark may contain multiple pages of extra code for initialization, parsing of command-line options, logging results, and so forth. Publishing only the benchmark core is not a valid solution, as the initialization code may contain subtleties that yield substantially different results.

This paper presents coNCePTuaL, a solution to the problem of benchmark opacity. coNCePTuaL—the capitalized letters represent "Network Correctness and Performance Testing Language"—is a high-level specification language designed specifically for testing the correctness and performance of communication networks.[1] Complete benchmarks—including initialization, command-line parsing, execution timing, statistics gathering, and data logging—can be expressed in comparatively few lines of code and can thereby be reproduced easily alongside performance tables and graphs. coNCePTuaL's grammar is English-like and highly readable even without prior exposure to the language or knowledge of the precise semantics. The coNCePTuaL compiler has a modular back end which is not tied to a single language or messaging layer; the compiler's internal representation is sufficiently high-level as to support arbitrary language/messaging layer combinations. Finally, the coNCePTuaL run-time system's logging facility writes to every log file a wealth of information about the benchmark's execution environment, facilitating reproducible experimentation and results. The net outcome is that coNCePTuaL enables a scientific approach to communication benchmarking not otherwise practical, convenient, or necessarily even possible. With coNCePTuaL, experimental setups are precisely specified, performance results are unambiguous, and even the review process is streamlined and simplified.

The remainder of the paper is organized as follows. In Section 2 we show how coNCePTuaL differs from previous works. Section 3 describes the coNCePTuaL language. We discuss the implementation of the coNCePTuaL compiler, run-time system, and associated tools in Section 4. In Section 5 we demonstrate that a program expressed in coNCePTuaL produces the same results as the corresponding program hand-coded in a general-purpose programming language and present performance results from a "real-world" use of coNCePTuaL. Finally, Section 6 draws some conclusions based upon the work described in this paper.

2. Related Work

coNCePTuaL is not the first domain-specific specification language designed for network testing but it is arguably the most sophisticated. Prior state of the art is epitomized by MITRE's Local Scheduler Executor (LSE) language [10]. LSE supports—among other constructs—counted loops, synchronous and asynchronous communication operations, faked "computation", and the ability to log performance data, as does coNCePTuaL. However, LSE and its counterparts lack coNCePTuaL's expressive power. Consider the synchronous pipe benchmark described in Monk, et al.'s report [10]. In that benchmark, each row of nodes repeatedly requests and receives data from each of the nodes in the previous row. LSE requires separate code for each node in the program, leading to large, difficult-to-maintain programs. The message size, number of iterations, compute time, and node topology are all hardwired into the program. Like an assembly language, LSE code is rarely written by hand. Rather, LSE programs are normally produced by ad hoc scripts. In contrast, coNCePTuaL programs describe communication from a global perspective.

[1] coNCePTuaL is pending approval for public release and will eventually be available from `http://www.c3.lanl.gov/~pakin/software/` under an open-source license.

0-7695-2132-0/04/$17.00 (C) 2004 IEEE

They can accept command-line arguments which enable the message size, number of iterations, compute time, and node topology to be determined dynamically at run time, as opposed to statically at program-development time. While each line of LSE code is terse and only partially understandable to one unfamiliar with the format, coNCePTuaL programs read almost like English prose. LSE's logging consists of simple timestamped operations, while coNCePTuaL can log the value of arbitrary expressions, making explicit the statistical operations performed over the complete set of values. In short, coNCePTuaL is a featureful, readable, high-level language, while the prior state of the art in network-testing languages is more akin to an assembly language.

Multiple implementations of communication benchmarks differ in subtle ways which may greatly impact the performance results. coNCePTuaL's approach to this problem is to make benchmarking methodology precise and explicit. An alternative approach is to define a suite of communication benchmarks, each with a fixed implementation and known way of reporting performance. With this approach, everyone familiar with the benchmark suite can understand what an ensuing performance graph represents. The standard-benchmark approach, represented by such suites as PMB [12] and SKaMPI [14] generally involves a set of benchmark codes and test harnesses. A user merely compiles and runs the benchmarks and reports the results in a prescribed manner. The key difference between the standard-benchmark and coNCePTuaL approaches is that the former enforces fair comparisons of results but limits those comparisons to a stock set of benchmarks and requires global agreement to use that set. coNCePTuaL, in contrast, enables unlimited benchmark variety at the cost of merely clarifying when unlike data are being compared rather then limiting comparisons to like data. In addition, the standard-benchmark approach typically targets a specific messaging layer (MPI in the case of PMB and SKaMPI) while coNCePTuaL programs are portable across messaging layers. The tradeoff is that coNCePTuaL cannot test features unique to a particular messaging layer such as noncontiguous datatypes in MPI or remote method invocations in Java/RMI. Nevertheless, many standard benchmarks could be rewritten in coNCePTuaL, combining the advantages of both approaches.

In addition to its ability to clarify the execution and results of standard communication benchmarks, coNCePTuaL can also be used for custom benchmarks that are developed and iteratively refined with the goal of gaining understanding into how a system performs, as opposed to merely demonstrating superior network performance to a competitor's offering. Section 5 elaborates on this point.

3. Language

coNCePTuaL is a domain-specific specification language. Although it can express complex communication patterns, coNCePTuaL is not a Turing machine and is therefore not capable of general computation. For brevity, coNCePTuaL's complete, formal grammar is omitted from this paper; the interested reader is referred to the coNCePTuaL user's manual [11] for a more thorough presentation of the language. Instead, we present the language as a series of annotated examples designed to showcase some of the key features of the language.

3.1. A Latency Benchmark

coNCePTuaL was designed to be easy to read. The aim of Section 3.1 is to demonstrate that coNCePTuaL code is easy to write, as well. This section shows, tutorial-style, how to construct a latency benchmark in coNCePTuaL. The goal of a latency benchmark is to measure the time it takes to send a message from one node to another. Because clusters generally lack globally synchronized clocks, a latency benchmark normally has Node A send a message to Node B, who then sends an equal-sized message back to Node A. Node A then reports half of the round-trip time as the latency, commonly measured in microseconds.

Listing 1 depicts a complete, but trivial, coNCePTuaL program that performs a single round-trip message send. In the listing, keywords are shown in boldface. The following are some points to note about the code:

- The program is very English-like. coNCePTuaL programs are intended to read much like a human would describe a benchmark to another human.
- The language is whitespace- and case-insensitive. For clarity, the listings in this paper are formatted in a semantically meaningful way.
- The program is comprised primarily of keywords. For one unfamiliar with coNCePTuaL, keywords are easier to understand than symbols.
- coNCePTuaL programs refer to "tasks" instead of "nodes". This is because the mapping of (software) tasks onto (hardware) nodes is external to coNCePTuaL and is handled by a third-party job launcher (e.g., `mpirun`).
- Task 0's sending of a 0-byte message to task 1 implicitly causes task 1 to receive a 0-byte message from task 0 (and vice versa in line 2).
- By default, sending and receiving are blocking calls, corresponding, for example, to `MPI_Send()` and `MPI_Recv()` when MPI [9] is used as the underlying messaging layer.

0-7695-2132-0/04/$17.00 (C) 2004 IEEE

- No data is logged; Listing 1 merely sends a message in each direction then terminates.

Listing 1. The beginnings of a latency benchmark

```
Task 0 sends a 0 byte message to task 1
    then
task 1 sends a 0 byte message to task 0.
```

Simplicity was a primary design goal. The language has no complex datatypes (e.g., arrays or structures), no user-defined functions, and no variables as such, although it does allow values to be let-bound to names within a scope. Nevertheless, experience has shown that a wide variety of network performance and correctness tests can be written with the features provided.

A logical next step towards evolving Listing 1 into a complete latency benchmark is to have the program perform a number of back-to-back ping-pongs and log the average latency (defined as half of the round-trip time) to a file. Listing 2 presents the next evolution of the latency benchmark. coNCePTuaL's **logs** statement specifies an expression to log and a corresponding description, which is used verbatim as a column header in the log file. (See Section 4 for more details about log files.) coNCePTuaL implicitly maintains an `elapsed_usecs` variable which measures elapsed time in microseconds. Listing 2 reports the arithmetic mean of the round-trip time but coNCePTuaL also provides functions for computing the median, harmonic mean, standard deviation, minimum, maximum, or sum of a set of data. The log file even indicates what function was used so that there is no ambiguity as to how the data were aggregated.

Listing 2. Mean of 1000 ping-pongs

```
For 1000 repetitions {
  task 0 resets its counters then
  task 0 sends a 0 byte message to task
      1 then
  task 1 sends a 0 byte message to task
      0 then
  task 0 logs the mean of elapsed_usecs
      /2 as "1/2 RTT (usecs)"
}
```

As in many programming languages, loop bodies consist of only a single statement but curly braces can be used to introduce compound statements. Hence the **for** loop in Listing 2 encompasses the following four statements. The phrase "**resets its counters**" tells coNCePTuaL to zero out `elapsed_usecs` and other such counters and to restart the clock.

Listings 1–2 send only zero-byte messages. The final incarnation of our latency test, Listing 3, performs the test using a variety of message sizes. For a more "professional" touch, it also accepts various parameters from the command line; it verifies that it was given at least two processors; it explicitly specifies the version of the coNCePTuaL language it expects for both forward and backward compatibility as the language evolves; and, it is commented for additional readability.

Listing 3. The coNCePTuaL equivalent of `mpi_latency.c`

```
# D. K. Panda's ping-pong latency test
    rewritten in coNCePTuaL

Require language version "0.5".

# Parse the command line.
reps is "Number of repetitions of each
    message size" and comes from "--reps" or
    "-r" with default 10000.
wups is "Number of warmup repetitions of
    each message size" and comes from
    "--warmups" or "-w" with default 10.
maxbytes is "Maximum number of bytes to
    transmit" and comes from "--maxbytes" or
    "-m" with default 1M.

# Ensure that we have a peer with whom to
    communicate.
Assert that "the latency test requires at
    least two tasks" with num_tasks>=2.

# Perform the benchmark.
For each msgsize in {0}, {1, 2, 4, ...,
    maxbytes} {
  all tasks synchronize then
  for reps repetitions plus wups warmup
      repetitions {
    task 0 resets its counters then
    task 0 sends a msgsize byte message
        to task 1 then
    task 1 sends a msgsize byte message
        to task 0 then
    task 0 logs the msgsize as "Bytes"
        and
                the mean of elapsed_usecs
                    /2 as "1/2 RTT (usecs
                    )"
  } then
  task 0 flushes the log
}
```

Some noteworthy details of Listing 3 include the following:

0-7695-2132-0/04/$17.00 (C) 2004 IEEE

- As shown in line 8, constants can accept suffixes, which act as multipliers. For example, 64K represents 65,536 (64×1024) and 5E6 represents 5000 (5×10^6).

- For convenience, coNCePTuaL includes the notion of "warmup repetitions" as an idiom in the language (line 16). Non-idempotent operations such as writing to the log file are suppressed during warmup repetitions.

- Line 23 forces coNCePTuaL to calculate the mean and write that to the log file. Without a log flush, the mean calculation would apply across all message sizes instead of being constrained to a single size.

- coNCePTuaL loops can use mathematical set notation (line 14) to describe the range of numbers that the loop variable accepts.

Regarding the final point, variables can iterate over each entry in a fully specified set (e.g., "{2, 13, 5, 5, 3, 8}") or over a partially specified arithmetic or geometric progression (e.g., "{1, 3, 5, ..., 77}"). The coNCePTuaL compiler automatically figures out the sequence. Sets can be spliced together by commas; the loop variable will iterate over each set in turn. Listing 3 separates out the "0" because "{0, 1, 2, 4, ..., 1M}" is neither an arithmetic nor a geometric progression.

At 24 lines—16 excluding comments and blank lines—the complete benchmark can fit on a single page with abundant room remaining for explanatory text and performance graphs, yet the benchmark is precise enough for a reader to understand exactly what was measured and how the results are calculated. The only thing missing is a description of the environment in which the benchmark ran; this is covered in Section 4.1.

3.2. Additional Language Features

Section 3.1 introduced some of the basic features of the coNCePTuaL language, including parameter declarations, assertions, two different **for** loops, set notation, compound statements, the **send** statement, and statements for producing log files. We now briefly describe some of the language constructs not previously encountered. A complete description of the language is presented in the coNCePTuaL user's guide [11].

Communication Constructs The **send** statement accepts a variety of parameters. Messages can be sent synchronously or asynchronously. They can recycle message buffers or use a different buffer for every invocation. Buffers can be aligned on arbitrary byte boundaries. Buffers can be "touched" before sending and/or after reception. The data can be verified with bit errors automatically tallied. Listing 4, an all-to-all validation test, demonstrates asynchronous transmission, aligned message buffers, and data verification. Note that coNCePTuaL automatically maintains the `bit_errors` variable. (The technique used to tally bit errors is described in Section 4.2.) Listing 4 also demonstrates a new **for** construct which runs for a given length of time rather than for a given number of iterations. In addition to point-to-point messaging, the language also supports multicast messages and barrier synchronization.

Listing 4. A network correctness test

```
1  # Ensure that every task can send to
       every other task.
2
3  Require language version "0.5".
4
5  msgsize is "Number of bytes each task
       sends" and comes from "--msgsize" or
       "-m" with default 1K.
6  testlen is "Number of minutes for which
       to run" and comes from "--duration" or
       "-d" with default 1.
7
8  Assert that "this program requires at
       least two tasks" with num_tasks>1.
9
10 For testlen minutes
11   For each ofs in {1, ..., num_tasks-1} {
12     all tasks src asynchronously send a
           msgsize byte page aligned message
            with verification to task (src+
           ofs) mod num_tasks then
13     all tasks await completion
14   }
15
16 All tasks log bit_errors as "Bit errors".
```

Non-Communicating Statements Although coNCePTuaL's focus is on communication there are a few statements that do not perform communication operations but rather operate locally. In addition to the previously presented **logs**, **flushes the log**, **resets its counters**, and **assert**, coNCePTuaL provides the following locally operating statements (among others):

- **computes for** "computes" in a tight spin-loop for a given length of time and **sleeps for** relinquishes the CPU for a given length of time.

- **touches** walks a memory region with a given stride, touching the data as it goes along. This is useful both for mimicking computation and for measuring cache/memory performance.

0-7695-2132-0/04/$17.00 (C) 2004 IEEE

- **outputs** writes a string or expression to the standard output device—useful for debug and progress messages.

Expressions In addition to the basic relational and arithmetic operators, coNCePTuaL provides an exponentiation operator, bitwise operators, and relational operators to test evenness, oddness, and the divisibility of one number by another. Noteworthy functions included in the run-time system include a function for returning the minimum number of bits required to represent an integer (**bits**) and a function for rounding a number to the nearest single-digit factor of an integral power of 10 (**factor10**). The run-time system also supports various topology operations that compute parents and children in *n*-ary and *k*-nomial trees and arbitrary offsets in 1-D, 2-D, and 3-D meshes and tori.

Sets of tasks can be specified in a variety of ways. We have already seen **task** followed by a constant (e.g., in Listing 3) and "**all tasks**"—with an optional variable declaration—as in Listing 4. Some additional forms include "**a random task**", which randomly selects a task; "**a random task other than** *x*", which randomly selects a task guaranteed not to be equal to *x*; **task** followed by an expression, which refers only to those tasks whose rank matches the expression; and, a "such that" ("|") variation, which both declares a variable and restricts it to a given set of tasks. For example, "**task** x|x >0 /\ x<num_tasks−2" operates only on numbers that are both greater than zero and less than the maximum rank minus 2.

In short, the coNCePTuaL language is rich in functionality yet highly legible. coNCePTuaL programs read like English, making them easy to understand even without knowledge of the underlying formal grammar. Comparatively few lines of code can produce a complete benchmark, making coNCePTuaL uniquely suitable for rapidly producing and refining special-purpose network correctness and performance tests.

4. Implementation

coNCePTuaL is implemented in Python using the SPARK little-language framework [1]. The structure of the coNCePTuaL compiler is largely undistinguished: a lexer converts coNCePTuaL source code into a token list; a parser converts the token list into an abstract syntax tree (AST); and, and a code generator converts the AST into low-level code (e.g., C) including calls to a messaging library (e.g., MPI). The coNCePTuaL compiler does have a couple of noteworthy features, however:

1. The coNCePTuaL lexer canonicalizes keyword variants such as **send**/**sends**, **message**/**messages**, and **a**/**an** into a uniform representation to permit programs to more closely resemble grammatically correct English.
2. Because each component of the compiler is a standalone module, multiple code-generator modules are possible. A compiler command-line option dynamically selects a particular module at compile time. Although only C + MPI output is currently implemented, new code generators (e.g., C + TCP, Fortran + OpenMP, Java + RMI, etc.) should be straightforward to produce.[2]

What makes the compiler's code generation unique is that the same coNCePTuaL source code can target any language/library for which a code-generator module exists. This enables fair comparisons of communication performance across languages/libraries.

The coNCePTuaL run-time system consists of a library written in C and invariant across any code generator that produces code capable of invoking C functions. The library—in fact, all of coNCePTuaL—is configured using the GNU Autotools [15] and builds properly on a variety of Unix-like systems (Linux, IRIX, Solaris, and Tru64), on a variety of architectures (IA-32, IA-64, MIPS-4, Alpha EV67, and SPARC v9), and with a variety of compilers (GNU, Intel, MIPSpro, HP, and Sun). The library provides routines for memory allocation, statistics reporting, random-number generation, interrupt handling, log-file manipulation, data verification, and various functions that are exported to coNCePTuaL programs. The library can take advantage of PAPI [2] or a variety of platform-specific mechanisms for acquiring performance information. It can process command-line arguments—both program-specified and internally generated—and automatically provides support for a "`--help`" option that outputs program-specific usage information.

4.1. Log-file Format

One of the most important responsibilities of the coNCePTuaL run-time system is to log measurement data to a file in a clear, consistent, informative, and easily parseable format. To this end, log files contain the following pieces of information, with the data format shown in brackets after each item.

- information about the execution environment [K:V]
- all environment variables and their values [K:V]
- the complete program source code [text]
- the program-specific measurement data [CSV]
- various timestamps and information about resource utilization [K:V]

[2] Approximately 60 Python object methods implement the complete language. Many of these are independent of the target language/library but the others do need to be rewritten for each new language or library.

0-7695-2132-0/04/$17.00 (C) 2004 IEEE

Log files are stored in a simple format that is both easy for a human to read and easy for a program to parse. Measurement data is stored in comma-separated value ("CSV") format, i.e., columns separated by commas, rows separated by newline characters, and column-header string surrounded by double quotes. All other log-file content—much in the form of simple ⟨*key*⟩:⟨*value*⟩ pairs ("K:V")—is considered commentary and is stored in lines beginning with "#".

coNCePTuaL logs a wealth of information about the execution environment. This includes information about the system architecture, operating system, library build environment, microsecond timer, and application-specific command-line parameters. It even logs warning messages if the microsecond timer exhibits poor granularity, a large standard deviation, or if it timer utilizes a 32-bit cycle counter and therefore wraps around every few seconds. The intention is that the log file present enough information to fully reproduce an experiment and gauge the validity of the reported results—an ability not readily possible without coNCePTuaL.

The data in a log file is written with two rows of column headings. The first row is the string provided to the **logs** statement. The second row describes how the column data were aggregated. For example, the column headers produced by Listing 3 are shown in Figure 2.

```
"Bytes","1/2 RTT (usecs)"
"(only value)","(mean)"
```

Figure 2. Log-file column headers associated with Listing 3

4.2. Verification

coNCePTuaL takes a unique approach to verifying messages sent "**with verification**". Rather than include with the message a CRC word, which has limited ability to report severe data corruption, the sender fills each message buffer with a random-number seed followed by the initial *N* random numbers generated using that seed. (The coNCePTuaL run-time system utilizes the Mersenne Twister [8] for its speed and randomness properties.) To verify the message contents, the receiver seeds its random-number generator with the first word of the message, generates *N* random numbers, and compares these to the message contents. coNCePTuaL is thus able to accurately report the total number of uncorrected bit errors that made it past the network and software stacks undetected.[3] This number is exported to a coNCePTuaL program as the variable `bit_errors` and can be logged like any other data.

[3]Exception: If a bit error corrupts the seed word, coNCePTuaL may report an artificially large number of bit errors.

4.3. Additional Tools

In addition to the compiler and run-time system, the coNCePTuaL system provides a few tools that are useful for presenting measurements made with coNCePTuaL. `logextract` is a Perl script that extracts various pieces of information from a log file and formats them for presentation or inclusion into another software package. Most importantly, `logextract` can discard the comments from a log file, extract the CSV data, and reformat it for immediate import by various spreadsheets or graphing packages. In addition, `logextract` can format the data or execution-environment information for presentation. For instance, `logextract` can extract the execution-environment information from a log file and format it using the LaTeX typesetting system. The coNCePTuaL system also includes syntax highlighters for a variety of editors and pretty-printers for a variety of formatting systems. (These are all generated automatically so they stay consistent with the language.) All of the code listings in this paper were produced using one of these pretty-printers. In summary, coNCePTuaL is a complete system for network correctness and performance testing, not just a single, standalone tool.

5. Evaluation

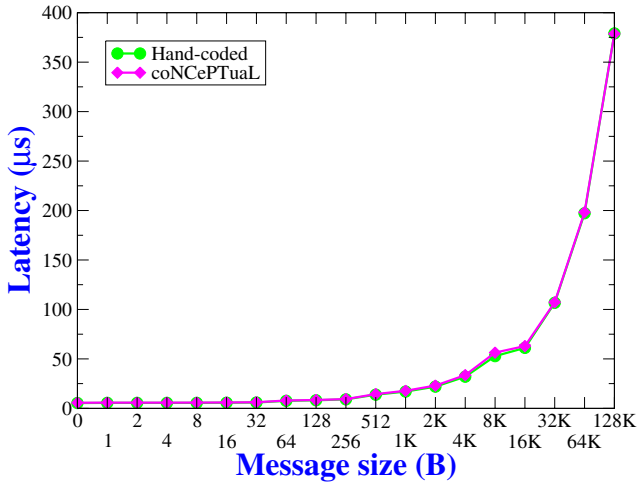
It is important for coNCePTuaL's success to demonstrate that programs written in coNCePTuaL produce the same results as the corresponding hand-coded program written using the target language and messaging layer. To demonstrate this, we faithfully converted the 58-line C + MPI latency test available from `http://nowlab.cis.ohio-state.edu/projects/mpi-iba/performance/mpi_latency.c` into the 16-line coNCePTuaL version previous shown in Listing 3 on page 4 and the 89-line C + MPI bandwidth test available from `http://nowlab.cis.ohio-state.edu/projects/mpi-iba/performance/mpi_bandwidth.c` into the 15-line coNCePTuaL version shown in Listing 5. (All line counts exclude blanks and comments.) These tests were chosen because they are typical communication microbenchmarks, they were written by a third party, and they have previously been used to compare performance across different types of networks [6]. Note that the coNCePTuaL versions actually improve upon the originals: the maximum message size is not hardwired into the program; the number of warmup iterations is read from the command line, not hardwired into the program; and, multiple messages sizes are tested per program invocation. Furthermore, the coNCePTuaL versions automatically log all of the items described in Section 4.1 in addition to the size and latency/bandwidth measurements.

Figure 3(a) compares the communication latencies measured by the hand-coded `mpi_latency` test with the coNCePTuaL equivalent presented in Listing 3. Figure 3(b)

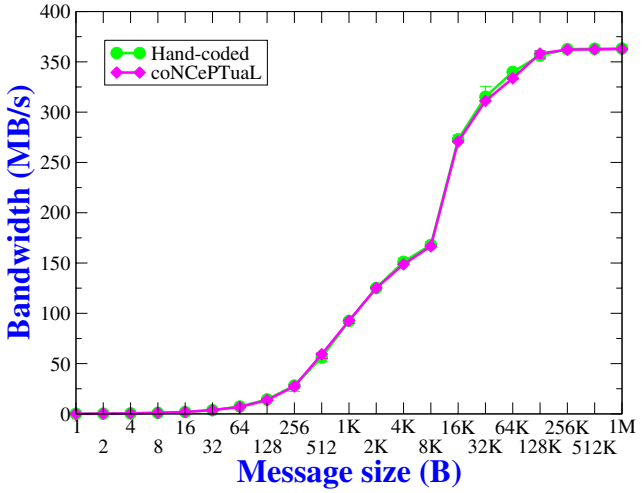
0-7695-2132-0/04/$17.00 (C) 2004 IEEE

Listing 5. The coNCePTuaL equivalent of `mpi_bandwidth.c`

```
# D. K. Panda's bandwidth test rewritten
    in coNCePTuaL

Require language version "0.5".

reps is "Number of repetitions of each
    message size" and comes from "--reps" or
     "-r" with default 1000.
maxbytes is "Maximum number of bytes to
    transmit" and comes from "--maxbytes" or
     "-m" with default 1M.

For each msgsize in {1, 2, 4, ...,
    maxbytes} {
  # Send some warm-up messages.
  task 0 asynchronously sends reps
      msgsize byte page aligned messages
      to task 1 then
  all tasks await completion then
  task 1 sends a 4 byte message to task
       0 then
  all tasks synchronize then
  # Perform the actual test.
  task 0 resets its counters then
  task 0 asynchronously sends reps
      msgsize byte page aligned messages
      to task 1 then
  all tasks await completion then
  task 1 sends a 4 byte message to task
       0 then
  task 0 logs msgsize as "Bytes" and
              bytes_sent/elapsed_usecs as
                  "Bandwidth"
}
```



(a) Latency



(b) Bandwidth

Figure 3. Hand-coded benchmarks vs. their coNCePTuaL equivalents

compares the communication bandwidth measured by the hand-coded `mpi_bandwidth` test with the coNCePTuaL equivalent presented in Listing 5. The data were measured on an Itanium 2 cluster interconnected with a Quadrics network [13]. As Figure 3 shows, there is no qualitative difference between the curves representing the coNCePTuaL and hand-coded versions of the programs. In short, the C + MPI code generated automatically by coNCePTuaL compares extremely favorably to its hand-coded equivalent.

Although the most popular use of communication benchmarks is to compare the performance of disparate networks on a variety of communication patterns, another important application is application-centric analytical performance modeling [3, 4, 7]. This form of modeling is commonly used to accurately predict the performance of a given application when run on a particular computer system or to assess if an application is observing the performance it could be expected to observe. The idea is to parameterize an application's performance in terms of system performance characteristics, such as CPU speed and network performance, which are quantified by benchmarks and plugged into the model to produce a performance prediction. These benchmarks tend to be application-specific and generally encapsulate performance achieved during an application's actual activities instead of the peak performance that could potentially be achieved by a carefully crafted program. The subset of those benchmarks which are used to measure communication performance parameters, being specific to a single

0-7695-2132-0/04/$17.00 (C) 2004 IEEE

application, usually have short lifetimes and should therefore ideally be simple to write. They may also represent complex communication patterns, making them difficult to express using a low-level language and messaging library. coNCePTuaL is therefore an ideal tool for generating the custom benchmarks needed for application-centric analytical performance modeling.

Consider, for example, the performance model used by Kerbyson et al. to accurately predict the performance of SAGE, a 150,000-line Eulerian hydrodynamics code that is representative of part of the ASCI workload [4]. One of the performance parameters used in that work represents communication latency and bandwidth in the presence of network contention. Listing 6 is a coNCePTuaL version of the network-contention benchmark used to gather performance data for Kerbyson et al.'s SAGE model. The benchmark measures ping-pong performance between task 0 and task $N/2$ first in isolation, then concurrently with repeated ping-pong communication between tasks 1 and $N/2 + 1$, then concurrently with that communication and with ping-pong communication between tasks 2 and $N/2 + 2$, and so forth. Figure 4 presents the results of running the SAGE network-contention program on a 16-processor SGI Altix 3000 NUMA system [16]. As the figure shows, performance drops immediately when going from no contention to a single competing ping-pong but drops no further when the contention level is increased. This indicates that the (2-CPU) front-side bus is the bandwidth bottleneck and that the remainder of the network has sufficient capacity to support eight concurrent ping-pongs.

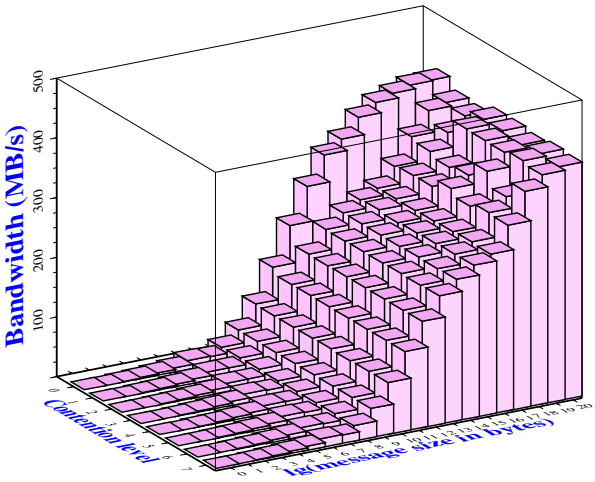


Figure 4. Network contention on a 16-processor Altix, as measured by coNCePTuaL

6. Conclusions

Although communication benchmarks are an important tool for evaluating network performance, a serious problem that this paper identifies is that performance measurements are frequently opaque. That is, only the benchmark's author knows exactly what the benchmark does and the precise conditions under which the measurements were taken. As a result, disparate data are wrongly compared, incorrect conclusions are drawn, and measurements are utterly irreproducible by third parties. Even special-purpose benchmarks not intended to be shared with others suffer from benchmark opacity in cases in which the author unearths old performance results but has no record of what benchmark produced those results, what parameters were utilized, or what computing environment was utilized.

We presented a solution to the problem of benchmark opacity in the form of coNCePTuaL, a new domain-specific language designed primarily to improve communication-benchmarking methodology. coNCePTuaL's most noteworthy features are that (1) the coNCePTuaL language is English-like and easily readable by people who know nothing about coNCePTuaL; (2) log files produced by the coNCePTuaL run-time system include not just the performance results but also a wealth of information about the execution environment and the complete benchmark source code; and, (3) back-end code generated by the coNCePTuaL compiler yields nearly identical performance results to the hand-written, hand-optimized version of the same benchmark. In addition to network performance benchmarking, coNCePTuaL supports network correctness testing. It supports a novel technique for tallying bit errors that pass undetected through the network hardware and messaging software, making it possible to accurately gauge a cluster's fault rate or error-correction efficacy. Finally, coNCePTuaL is well-suited for producing the one-of-a-kind benchmarks that lie at the core of application-centric analytical performance modeling and are designed to provide insight into application performance.

The primary conclusion that one should draw from this work is by combining a domain-specific language, a modular compiler, and a rich run-time system, coNCePTuaL makes it possible to present network performance data in a form that can be easily understood and properly subjected to peer review. The result is that coNCePTuaL enables a more scientific approach to benchmarking network performance than is otherwise possible.

References

[1] J. Aycock. Compiling little languages in Python. In *Proceedings of the Seventh International Python Conference*, pages 69–77, Houston, Texas, Nov. 10–13, 1998.

[2] S. Browne, J. Dongarra, N. Garner, G. Ho, and P. Mucci. A portable programming interface for performance evaluation

0-7695-2132-0/04/$17.00 (C) 2004 IEEE

on modern processors. *The International Journal of High Performance Computing Applications*, 14(3):189–204, Fall 2000.

[3] A. Hoisie, O. Lubeck, and H. Wasserman. Performance and scalability analysis of teraflop-scale parallel architectures using multidimensional wavefront applications. *The International Journal of High Performance Computing Applications*, 14(4), Nov. 2000.

[4] D. J. Kerbyson, H. J. Alme, A. Hoisie, F. Petrini, H. J. Wasserman, and M. Gittings. Predictive performance and scalability modeling of a large-scale application. In *Proceedings of SC2001*, Denver, Colorado, Nov. 10–16, 2001.

[5] D. J. Kerbyson, A. Hoisie, and H. J. Wasserman. A comparison between the Earth Simulator and AlphaServer systems using predictive application performance models. In *Proceedings of the 2003 International Parallel and Distributed Processing Symposium (IPDPS)*, Nice, France, Apr. 22–26, 2003.

[6] J. Liu, J. Wu, S. P. Kinis, D. Buntinas, W. Yu, B. Chandrasekaran, R. Noronha, P. Wyckoff, and D. K. Panda. MPI over InfiniBand: Early experiences. Technical Report OSU-CISRC-10/02-TR25, Computer and Information Science Department, The Ohio State University, Columbus, Ohio, Jan. 30, 2003.

[7] M. Mathis, D. J. Kerbyson, and A. Hoisie. A performance model of non-deterministic particle transport on large-scale systems. In P. M. A. Sloot, D. Abramson, A. V. Bogdanov, J. J. Dongarra, A. Y. Zomaya, and Y. E. Gorbachev, editors, *Proceedings of the International Conference on Computational Science (ICCS), Part III*, volume 2659 of *Lecture Notes in Computer Science*, pages 905–915, Melbourne, Australia and St. Petersburg, Russia, June 2–4, 2003.

[8] M. Matsumoto and T. Nishimura. Mersenne Twister: A 623-dimensionally equidistributed uniform pseudorandom number generator. *ACM Transactions on Modeling and Computer Simulations*, 8(1):3–30, Jan. 1998.

[9] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard*, June 12, 1995.

[10] L. Monk, R. Games, J. Ramsdell, A. Kanevsky, C. Brown, and P. Lee. Real-time communications scheduling: Final report. Technical Report MTR 97B0000069, The MITRE Corporation, Bedford, Massachusetts, May 1997.

[11] S. Pakin. coNCePTuaL user's guide. Los Alamos Unclassified Report 03-7356, Los Alamos National Laboratory, Los Alamos, New Mexico, Oct. 2003.

[12] Pallas, GmbH. *Pallas MPI Benchmarks—PMB, Part MPI-1*, Mar. 9, 2000.

[13] F. Petrini, W. Feng, A. Hoisie, S. Coll, and E. Frachtenberg. The Quadrics network (QsNet): High-performance clustering technology. *IEEE Micro*, 22(1):46–57, Jan.–Feb. 2002.

[14] R. H. Reussner. SKaMPI: The special Karlsruher MPI-benchmark user manual. Technical Report 99/02, Department of Informatics, Universität Karlsruhe, Dec. 3, 2002.

[15] G. V. Vaughan, B. Elliston, T. Tromey, and I. L. Taylor. *GNU Autoconf, Automake, and Libtool*. New Riders, Indianapolis, Indiana, Oct. 6, 2000.

[16] M. Woodacre, D. Robb, D. Roe, and K. Feind. The SGI Altix 3000 global shared-memory architecture. White paper, SGI, Mountain View, California, Apr. 30, 2003.

Listing 6. Bandwidth in the presence of network contention

```
# Measure the intratask network contention factor as used by the
# analytical SAGE performance model
#
# Benchmark by Darren J. Kerbyson
# Implementation in coNCePTuaL by Scott Pakin

Require language version "0.5".

reps is "number of repetitions" and comes from "--reps" or "-r" with default 1000.
minsize is "minimum message size" and comes from "--minsize" or "-m" with default 0.
maxsize is "maximum message size" and comes from "--maxsize" or "-x" with default 1M.

Assert that "the number of tasks must be even" with num_tasks is even.

For each j in {0, ..., num_tasks/2-1} {
  task 0 outputs "Working on contention factor " and j then
  for each msgsize in {maxsize, maxsize/2, maxsize/4, ..., minsize} {
    all tasks synchronize then
    task 0 resets its counters then
    for reps repetitions {
      task i|i<=j sends a msgsize byte message to task i+num_tasks/2 then
      task i|i>j sends a msgsize byte message to task i-num_tasks/2
    } then
    task 0 logs j as "Contention level" and
                msgsize as "Msg. size (B)" and
                elapsed_usecs/(2*reps) as "1/2 RTT (us)" and
                (1E6*msgsize*2*reps)/(1M*elapsed_usecs) as "MB/s"
  }
}
```


0-7695-2132-0/04/$17.00 (C) 2004 IEEE